BOOK FOUR · THE AI ECONOMY MONETIZATION SERIES

The Monetization Protocol

for the AI Economy

The rules, objects, APIs, and agent interfaces governing how value is priced, metered, billed, and settled across the AI stack

TCP/IP defined the internet's communication layer. The Monetization Protocol defines the AI economy's commercial layer.

PREFACE

Why the AI Economy Needs a Monetization Protocol

The integration tax. The semantic inconsistency. The agent incompatibility. Three solvable problems — if we agree on the rules.

Every major technological platform shift in history eventually produces a protocol — a shared set of rules that makes the platform's parts interoperable. The internet needed TCP/IP before it could scale. The web needed HTTP before it could connect. Email needed SMTP before it could route. Electronic payments needed ISO 8583 before they could clear across banks that had never met.

The AI economy needs a monetization protocol.

Not because the current situation is unworkable — it is. Hundreds of companies have built their own proprietary representations of products, prices, contracts, entitlements, metering events, invoices, and credits. Some of those representations are sophisticated. Some are cobbled together from spreadsheets and webhook callbacks. None of them talk to each other. An enterprise trying to manage AI spending across five vendors is managing five different data models, five different APIs, five different interpretations of what an 'entitlement' means, and five different invoice formats that bear no resemblance to each other. The accounting team reconciles this manually. The finance team forecasts it by feel. The FinOps team watches dashboards that show spend but not value. The CFO approves budgets without reliable data.

This is not a sustainable state for an economy that is about to become the largest commercial infrastructure in human history.

The Monetization Protocol is my attempt to define that shared set of rules. It is not a standard in the formal sense — I am not the IEEE, and this is not an RFC. But it is a rigorous proposal: a complete specification of the objects, events, APIs, and agent interfaces that would allow any two participants in the AI economy to transact commercially with confidence, accuracy, and mutual intelligibility. It is the architecture I believe the industry needs to converge on. And it is the reference specification that every practitioner in the AI economy — every platform architect, every agent developer, every RevOps engineer, every finance leader — needs to understand before the convergence happens without them.

This book is divided into five parts. Part One establishes why a protocol is needed and what it must accomplish. Part Two defines the object model — the thirteen canonical data objects and their precise schemas. Part Three specifies the events — the atomic signals of commercial activity in the AI economy. Part Four defines the APIs — the standard interfaces through which any system can price, meter, bill, and settle. Part Five defines the agent interfaces — how AI agents participate in the commercial economy as buyers, sellers, and brokers.

A sixth part provides the MCP (Model Context Protocol) server specification: a complete implementation of the monetization protocol as an MCP server, with all tools, resources, and prompts defined, so that any AI agent with MCP capability can immediately participate in the monetization protocol without custom integration.

The protocol is not complete. No protocol ever is at first publication. The goal is not perfection — it is to establish a coherent foundation that others can build on, debate, and improve. The AI economy is being designed right now. The commercial layer of that economy needs to be designed with the same care and rigor as its technical layers. This book is a contribution to that design.

PART ONE

The Case for a Protocol

Why the AI economy's commercial infrastructure is failing and what a protocol must accomplish to fix it.

CHAPTER ONE

The Protocol Gap

Every major platform shift produces a protocol. The AI economy's commercial layer has not produced one yet — and the cost of that absence is compounding.

The history of internet infrastructure is a history of protocols preceding scale. This is not a coincidence. Scale requires standardization. Standardization requires protocols. And protocols require someone to sit down and define, with unusual precision, what the things are called, how they are represented, and how they move from one system to another.

In 1974, Vint Cerf and Bob Kahn published "A Protocol for Packet Network Intercommunication" — the paper that became the foundation of TCP/IP. The internet did not exist yet in any meaningful commercial sense. But the protocol they defined made it possible for the internet to exist, because it specified the rules that allowed fundamentally different networks to exchange data as if they were the same network. The genius of TCP/IP was not in what it did — it moved packets from one place to another, which was not conceptually revolutionary. The genius was in the precision with which it defined the rules: exactly how a packet was structured, exactly how addressing worked, exactly what happened when a packet was lost. That precision made interoperability possible. Interoperability made scale possible. Scale made everything else possible.

The web's HTTP protocol, published by Tim Berners-Lee in 1991, did the same thing at a higher layer. It defined how a client and a server should speak to each other: exactly what a request looked like, exactly what a response looked like, exactly what the status codes meant. Any HTTP client could speak to any HTTP server. This universality was the precondition for the web's explosion. Without a shared protocol at the application layer, the web would have fragmented into incompatible proprietary networks — the fate that befell early commercial online services like CompuServe and AOL.

The AI economy is approaching the same inflection point. The commercial infrastructure of the AI economy — the systems that price AI services, meter their consumption, issue invoices, process payments, recognize revenue, and govern spending — is currently a collection of proprietary, incompatible implementations. Every major AI vendor has its own data model. Every enterprise AI deployment has its own metering architecture. Every billing system represents entitlements, credits, and allocations differently. The result is a commercial layer that functions, barely, through manual integration work, but that cannot scale to the transaction volumes the AI economy will eventually require.

The problem is visible in three specific failure modes that practitioners encounter regularly. The first is integration tax: the cost of connecting AI commercial systems that

were not designed to interoperate. An enterprise that deploys AI from three vendors spends significant engineering resources building custom integrations between each vendor's billing API and the enterprise's financial systems — integrations that break when vendors update their APIs, that require maintenance as data models evolve, and that must be rebuilt for each new vendor. This integration tax is not a feature of AI commerce. It is a symptom of the absence of a shared protocol.

The second failure mode is semantic inconsistency: the same commercial concept represented differently across systems, making aggregation and comparison impossible. What one vendor calls an 'entitlement', another calls a 'quota' and a third calls a 'license seat'. What one system records as a 'usage event', another records as a 'consumption log entry' and a third records as a 'billing transaction'. When the finance team tries to build a unified view of AI spending, they are not comparing numbers — they are first translating between incompatible vocabularies, then comparing numbers. The translation is expensive, error-prone, and never quite complete.

The third failure mode is agent incompatibility: the emerging reality that AI agents are increasingly acting as commercial participants — purchasing services, negotiating terms, processing payments — but doing so without any shared language for commercial interaction. An AI agent that wants to purchase a service from another AI agent currently has no standard way to discover that service's pricing, no standard format for expressing the terms of a transaction, no standard mechanism for verifying that the service was delivered, and no standard protocol for settling the payment. Each agent-to-agent commerce implementation is a bespoke bilateral agreement, unscalable to the volume of machine-to-machine transactions the AI economy will generate.

The Monetization Protocol addresses all three failure modes by providing what every successful commercial protocol has provided: a shared vocabulary, a shared data model, and shared interfaces that allow any two participants to transact with mutual intelligibility.

> ***"TCP/IP defined the internet's communication layer. HTTP defined the web's transfer layer. The Monetization Protocol defines the AI economy's commercial layer."***

What a Protocol Is — and What It Is Not

A monetization protocol is not a software product. It is not a platform, a service, or an API managed by a single vendor. It is a specification — a document that defines the rules with sufficient precision that any developer, in any organization, using any programming language, can implement the rules independently and produce a system that is interoperable with any other implementation of the same rules.

This distinction is critical and easily misunderstood. When practitioners first encounter the concept of a monetization protocol, they often ask: which company will own it? The answer is: no company should own it, any more than any company owns TCP/IP or HTTP. A protocol gains its power precisely from the fact that it is neutral — that no single participant controls it, and therefore every participant can trust it as a foundation.

The Monetization Protocol specification in this book is proposed as an open standard. Its schemas, its event taxonomy, its API definitions, and its agent interface specifications are published without restriction. Any company implementing these specifications produces a monetization system that is interoperable with any other company implementing the same specifications. The protocol is the agreement. The implementations are the participants.

What the protocol defines specifically: first, the canonical objects — the thirteen data structures that represent the fundamental entities in AI commercial activity, from the Product object that defines what is being sold to the Outcome object that records what was delivered. Second, the canonical events — the standardized signals that record every commercially significant activity in an AI deployment, from a token being consumed to an agent task being completed. Third, the standard APIs — the interfaces through which

systems price, meter, bill, and settle. Fourth, the agent interfaces — the mechanisms through which AI agents participate in commercial activity as autonomous economic actors. Fifth, the governance rules — the dispute resolution procedures, the audit trail requirements, and the compliance frameworks that make the protocol trustworthy.

What the protocol does not define: implementation details, vendor-specific extensions, or any aspect of commercial strategy. The protocol specifies what a Product object must contain; it does not specify what products any company should offer. The protocol specifies how a metering event should be structured; it does not specify what pricing model any company should apply. The protocol is a language, not a policy.

Three Protocol Precedents Worth Studying

Three protocol precedents are worth understanding in detail because they illuminate both the design principles and the adoption dynamics that a monetization protocol must navigate.

The Financial Information eXchange (FIX) protocol, developed in 1992 by Fidelity Investments and Salomon Brothers, standardized electronic trading communications between financial institutions. Before FIX, each pair of institutions that wanted to trade electronically had to negotiate their own proprietary communication format. After FIX, any institution implementing the FIX protocol could trade electronically with any other FIX-compliant institution. The protocol was not mandated by regulation. It was adopted because it reduced the bilateral integration tax dramatically — it was cheaper to implement one standard than to maintain dozens of proprietary connections. FIX is now used for trillions of dollars of securities trades daily.

The Open Banking standard, developed under the UK's PSD2 regulation and adopted globally, standardized the APIs through which third-party applications could access bank account data and initiate payments. Before Open Banking, a fintech company that wanted to access customer account data needed individual agreements with each bank, each with its own API format and authentication mechanism. After Open Banking, any

compliant fintech could access any compliant bank's data through a standard API. The protocol unlocked an entire ecosystem of financial applications that could not have existed without the standard.

The Electronic Data Interchange (EDI) standards, developed through the 1970s and 1980s and codified as ANSI X12, standardized the electronic exchange of business documents — purchase orders, invoices, shipping notices — between trading partners. EDI adoption created the electronic supply chain and enabled the retail revolution of the 1990s. Walmart's legendary supply chain efficiency was built on EDI. The standard made it possible for a retail giant to have real-time inventory visibility across thousands of suppliers without custom bilateral integrations with each one.

The common thread across all three precedents is adoption economics: protocols spread when the cost of implementing the standard is lower than the cost of the bilateral integration tax it replaces. The Monetization Protocol will spread by the same mechanism. The AI economy's integration tax is already high and growing with every new vendor, every new deployment, and every new agent capability. The moment a critical mass of vendors and enterprises implement the protocol, the economics of adoption become compelling for everyone who has not yet implemented it.

The Three Protocol Precedents — Design Principles and Adoption Patterns

Field	Type / Values	Description
FIX Protocol	Financial trading · 1992	Standardized electronic trading between financial institutions. Adopted because bilateral integration cost exceeded standardization cost. Now processes trillions daily.
Open Banking	Banking APIs · PSD2 · 2016	Standardized APIs for account data access and payment initiation. Unlocked fintech ecosystem. Adoption driven by regulation + economics.
EDI / ANSI X12	Supply chain · 1970s–80s	Standardized electronic business documents. Enabled Walmart's supply chain revolution. Pure economics — cheaper than bilateral integration.
Monetization Protocol	AI commerce · 2025	Standardizes AI commercial objects, events, APIs, agent interfaces. Adoption economics: integration tax already exceeds standardization cost.

PART TWO

The Object Model

Thirteen canonical objects. The shared vocabulary of AI commerce.

Before we can specify a protocol, we need to understand precisely what problem it solves, what it means for a protocol to exist in commercial AI infrastructure, and why the absence of one creates compounding costs that are not immediately visible but are ultimately unsustainable.

The object model is the foundation of everything else in the protocol. Before we can define events, APIs, or agent interfaces, we need to define the things those events, APIs, and interfaces operate on. The thirteen canonical objects are the vocabulary of AI commerce. Every commercial activity in the AI economy can be described using these objects and the relationships between them.

The design principles for the object model are strict. Every object has a unique identifier that is globally unique across all implementations. Every object has a defined set of required fields and a defined set of optional fields. Every object has a defined set of lifecycle states and the valid transitions between those states. Every object has a defined set of relationships to other objects, with cardinality and referential integrity constraints. Every object has a defined event history — a list of the events that can affect its state. And every object has a defined representation in JSON and in a wire format suitable for API transmission.

These constraints are not bureaucratic overhead. They are what makes interoperability possible. A Product object that is precisely defined can be created in one system and consumed by another system that has never interacted with the first system. The precision is the protocol.

CHAPTER TWO

The Canonical Objects

Part 2a — Products, Prices, Entitlements, Meters, and Events

Object 1 — Product

The Product object is the commercial representation of an AI capability being offered for sale. It is the top of the object hierarchy — every other commercial activity derives ultimately from a Product.

Required fields: id (UUID, globally unique), name (string, human-readable display name), type (enum: token_bundle | agent_workflow | outcome_service | model_access | compute_reservation | platform_subscription | composite), status (enum: draft | active | deprecated | archived), created_at (ISO 8601 timestamp), version (semantic version string).

Optional fields: description (string), tags (array of strings), parent_product_id (UUID, for composite products), ai_layer (enum: compute | model | token | agent | outcome), model_ids (array of UUIDs, the foundation models this product uses), capability_description (string, natural language description of what the AI can do), pricing_reference (UUID, reference to the primary Price object for this product).

Lifecycle states: draft (product is being configured, not available for sale), active (product is available for sale and consumption), deprecated (product is no longer available for new sales but existing entitlements continue), archived (product is fully retired, all entitlements have expired).

The type field deserves extended treatment because it captures the layer of the AI economy at which the product operates, which determines which pricing models and entitlement structures are valid for it. A token_bundle product is priced per token consumed and governed by a Token Budget entitlement. An agent_workflow product is priced per task or workflow completion and governed by a Task Limit entitlement. An

outcome_service product is priced per verified business outcome and governed by an Outcome SLA entitlement. A model_access product is priced per API call or per million tokens and governed by a Rate Limit entitlement. A compute_reservation product is priced per unit of reserved capacity and governed by a Capacity Reservation entitlement. A platform_subscription product is priced per user or per period and governed by a Seat entitlement.

The composite type is the most powerful and the most complex: it allows multiple product types to be bundled into a single commercial offering with a single price, a unified entitlement structure, and coherent revenue recognition across all component products. Most enterprise AI products are composites in practice, even when they are not represented as such in current billing systems.

Product Object — Required Fields

Field	Type / Values	Description
`id`	UUID v4	Globally unique identifier. Generated at creation. Never reused.
`name`	string	Human-readable display name. Maximum 200 characters.
`type`	enum	token_bundle | agent_workflow | outcome_service | model_access | compute_reservation | platform_subscription | composite
`status`	enum	draft | active | deprecated | archived
`created_at`	ISO 8601	Creation timestamp with timezone. Immutable after creation.
`version`	semver	Semantic version string. Incremented on any substantive change to product attributes.

Product Type → Valid Pricing Models and Entitlement Structures

Field	Type / Values	Description
`token_bundle`	Per-token Price	Token Budget entitlement · Meter type: token · Reset: monthly or contract_period
`agent_workflow`	Per-task Price	Task Limit entitlement · Meter type: task · Completion signal required

outcome_service	Per-outcome Price	Outcome SLA entitlement · Meter type: outcome · Verification method required
model_access	Per-token or API-call Price	Rate Limit entitlement · Meter type: token or api_call
compute_reservation	Reservation Price	Capacity Reservation entitlement · Meter type: time
platform_subscription	Subscription Price	Seat entitlement · Meter type: user or flat
composite	Hybrid Price (components array)	Multiple entitlements · Multiple meters · Allocation object required for rev rec

Object 2 — Price

The Price object defines the commercial terms under which a Product is sold. The relationship between Product and Price is one-to-many: a single Product may have multiple active Price objects simultaneously, representing different pricing tiers, different market segments, different geographic regions, or different contract vintages.

Required fields: id (UUID), product_id (UUID, reference to the Product this price applies to), name (string), type (enum: per_token | per_task | per_outcome | subscription | reservation | revenue_share | hybrid), status (enum: draft | active | deprecated), currency (ISO 4217 currency code), effective_from (ISO 8601 date), created_at (ISO 8601 timestamp).

Type-specific required fields: for per_token — input_token_price (decimal, price per million input tokens), output_token_price (decimal, price per million output tokens); for per_task — task_price (decimal, price per task completion), task_definition (string, precise definition of what constitutes a completed task); for per_outcome — outcome_price (decimal, price per verified outcome), outcome_definition (string, precise definition of what constitutes a verified outcome), verification_method (string); for subscription — period (enum: monthly | annual | multi_year), seats_included (integer, 0 for unlimited), overage_price (optional, decimal per unit above included allocation); for reservation — unit (enum: gpu_hour | instance_hour | token_bucket),

quantity (integer), period (enum: monthly | annual), reservation_price (decimal per unit per period).

Optional fields: volume_tiers (array of tier objects, each with min_quantity, max_quantity, and price), discount_percentage (decimal, for promotional or negotiated discounts), minimum_commitment (decimal, minimum spend per period), trial_configuration (object, defining trial terms and duration), geographic_restrictions (array of ISO 3166 country codes), channel_restrictions (array of enum values: direct | marketplace | partner | self_serve).

The hybrid type requires special treatment. A hybrid price allows multiple pricing dimensions to be active simultaneously — for example, a monthly subscription that includes a token allocation with per-token overage pricing above that allocation, plus per-outcome charges for outcomes delivered. The hybrid price object contains a components array, each element of which is a complete price specification for one dimension of the hybrid. Revenue recognition for hybrid prices requires the allocation rules defined in the Allocation object.

Price Object — Core Required Fields

Field	Type / Values	Description
`id`	UUID v4	Globally unique. Multiple prices can reference the same product.
`product_id`	UUID	Reference to the product this price governs. Validated against Product object.
`type`	enum	per_token | per_task | per_outcome | subscription | reservation | revenue_share | hybrid
`currency`	ISO 4217	Three-letter currency code. e.g. USD, EUR, GBP.
`effective_from`	ISO 8601 date	Date from which this price applies. Historical prices cannot be deleted.
`status`	enum	draft | active | deprecated — active prices cannot be deleted, only deprecated.

Price Type — Additional Required Fields

Field	Type / Values	Description
per_token	input_token_price, output_token_price	Decimal, price per million tokens. output typically 2–4× input.
per_task	task_price, task_definition	task_definition is a precise string specifying exactly what constitutes a completed billable task.
per_outcome	outcome_price, outcome_definition, verification_method	outcome_definition must be precise enough that both parties can agree on whether it occurred.
subscription	period, seats_included, overage_price (optional)	Period: monthly | annual | multi_year. seats_included: 0 = unlimited.
reservation	unit, quantity, period, reservation_price	unit: gpu_hour | instance_hour | token_bucket.
hybrid	components (array of Price specs)	Each component is a full Price specification. Allocation rules determine billing.

Object 3 — Entitlement

The Entitlement object is the commercial permission — the record of what a specific customer has the right to consume, under what conditions, for what period. It is the link between the commercial agreement (the Contract) and the enforcement of consumption limits (the Meter).

Required fields: id (UUID), customer_id (UUID), contract_id (UUID), product_id (UUID), price_id (UUID), status (enum: pending_activation | active | suspended | expired | cancelled), activated_at (ISO 8601 timestamp, null if not yet activated), expires_at (ISO 8601 timestamp, null for perpetual), created_at (ISO 8601 timestamp).

Type-specific fields (determined by the Product's type): for token_bundle — token_budget (integer, total tokens allocated), tokens_consumed (integer, running total, updated by Meter), tokens_remaining (integer, computed), reset_policy (enum: none | monthly | annual | rolling_30d); for agent_workflow — task_limit (integer, maximum tasks per period), tasks_completed (integer, running total), period (enum: monthly | annual), concurrent_agent_limit (integer); for outcome_service —

sla_definition (object), outcomes_committed (integer, per period), outcomes_delivered (integer, running total), sla_breach_credits (decimal, accumulated credits for SLA failures); for compute_reservation — reserved_capacity (object with unit and quantity), utilization_percentage (decimal, computed), overage_enabled (boolean).

Lifecycle events that affect Entitlement state: entitlement.activated (on provisioning completion), entitlement.suspended (on payment failure or policy violation), entitlement.budget_warning (when consumption reaches 80% of budget), entitlement.budget_exhausted (when consumption reaches 100% of budget), entitlement.renewed (on automatic renewal), entitlement.cancelled (on customer cancellation), entitlement.amended (on contract modification that changes entitlement terms).

The Entitlement object is the most operationally critical object in the protocol because it governs real-time access control. When an AI system receives a request, it must check the relevant Entitlement before serving the request: is the entitlement active? Does the customer have remaining budget? Is the request within the permitted scope? The Entitlement is the enforcement point. If it is wrong — if it reflects a contract that has been amended without updating the entitlement, or a budget that has been reset without clearing the consumption counter — the entire consumption governance system fails.

Entitlement Lifecycle — Valid State Transitions

Field	Type / Values	Description
`pending_activation → active`	Triggered by: provisioning.completed event	Precondition: contract.activated, downstream systems provisioned
`active → suspended`	Triggered by: payment.failed (after grace period) or policy_violation	Effect: entitlement.check returns permitted=false
`suspended → active`	Triggered by: payment.received or policy_violation.resolved	Effect: entitlement.check resumes normal behavior

`active → expired`	Triggered by: entitlement expiry date reached	Effect: entitlement archived, Asset status → pending_renewal
`active → cancelled`	Triggered by: contract.terminated or customer cancellation	Effect: consumption recording stops, Asset status → expired
`active → amended`	Triggered by: contract.amended affecting this entitlement	Effect: new entitlement created, original archived with amendment reference

Object 4 — Meter

The Meter object defines how consumption of an AI product is measured. It is the specification, not the measurement itself — the rules by which raw usage signals are translated into billable quantities. The Meter is to the Monetization Protocol what the electricity meter is to the utility company: the agreed-upon instrument of measurement whose readings become the basis for billing.

Required fields: id (UUID), product_id (UUID), name (string), type (enum: token | task | outcome | time | api_call | composite), unit (string, human-readable unit of measurement such as "tokens", "tasks", "hours"), precision (integer, decimal places to track), aggregation_method (enum: sum | max | average | percentile_95), reset_period (enum: never | hourly | daily | monthly | annually | contract_period), created_at (ISO 8601 timestamp).

Type-specific fields: for token meters — input_multiplier (decimal, typically 1.0 for input tokens), output_multiplier (decimal, typically 2.0-4.0 for output tokens, reflecting higher compute cost), cache_multiplier (decimal, typically 0.1 for cached input tokens); for task meters — task_definition (string, precise specification of what constitutes a metered task), completion_signal (string, the system event that triggers task completion recording), partial_task_handling (enum: ignore | pro_rate | full_charge); for outcome meters — outcome_definition (string), verification_source (string, the system or process that verifies outcome delivery), verification_latency_hours (integer, maximum hours between delivery and verification).

The composite meter type is required for products that are billed on multiple dimensions simultaneously. A composite meter contains a components array, each element of which is a complete meter specification for one dimension, plus a combination_rule that specifies how the components are aggregated into a single billable quantity. For example, an agent workflow product might be billed per task plus per token consumed per task, requiring a composite meter that tracks both dimensions and generates a combined bill.

Meter Object — Field Reference

Field	Type / Values	Description
`type`	enum	token | task | outcome | time | api_call | composite
`unit`	string	Human-readable unit: 'tokens', 'tasks', 'hours', 'API calls'
`precision`	integer	Decimal places to track. 0 for integer quantities, up to 6 for fractional.
`aggregation_method`	enum	sum (default) | max | average | percentile_95
`reset_period`	enum	never | hourly | daily | monthly | annually | contract_period
`input_multiplier`	decimal	Token meters: multiplier applied to input token counts. Default: 1.0
`output_multiplier`	decimal	Token meters: multiplier applied to output token counts. Default: 1.0 (vendors typically set 2.0–4.0)
`cache_multiplier`	decimal	Token meters: multiplier for cached input tokens. Default: 0.1 (reflects compute savings)
`task_definition`	string	Task meters: precise specification of what constitutes a metered task completion
`completion_signal`	string	Task meters: the system event name that triggers task completion recording
`outcome_definition`	string	Outcome meters: the agreed definition of a verified outcome
`verification_source`	string	Outcome meters: system or process that provides outcome verification

Object 5 — Event

The Event object is the atomic record of a single commercially significant occurrence in an AI deployment. It is the rawest layer of commercial data — the equivalent of a log entry, but structured according to the protocol's canonical schema so that it can be processed, attributed, and billed by any compliant system.

Required fields: id (UUID, v4, generated at event creation), type (string, the event type from the canonical event taxonomy), timestamp (ISO 8601 timestamp with millisecond precision), source_system (string, identifier of the system that generated the event), customer_id (UUID), product_id (UUID), entitlement_id (UUID).

Optional but strongly recommended fields: session_id (UUID, groups related events from a single user interaction or agent workflow), agent_id (UUID, the AI agent that generated this event, if applicable), model_id (UUID, the foundation model used), workflow_id (UUID, for agent workflow events), correlation_id (string, for tracing events through multiple systems).

Type-specific payload fields (the payload is a JSON object whose schema is defined by the event type): for token.consumed — input_tokens (integer), output_tokens (integer), cached_input_tokens (integer, tokens served from cache), model_version (string); for task.completed — task_definition_id (UUID), task_duration_ms (integer), task_success (boolean), error_code (string, if task_success is false); for outcome.verified — outcome_definition_id (UUID), outcome_value (decimal, economic value of the outcome if calculable), verification_method (string), verification_source (string); for agent.action — action_type (string), action_target (string), action_result (string).

The Event object has exactly one immutable field: the id. Once an event has been created and written to the event store, its id cannot change and its payload cannot be modified. All corrections to events are made through Adjustment events that reference the original event's id. This immutability is the foundation of the audit trail — it ensures that the historical record of commercial activity cannot be retroactively altered.

Event Object — Core Fields		
Field	**Type / Values**	**Description**
id	UUID v4	Globally unique. Immutable after creation. Used as idempotency key.
type	string	Event type from the canonical taxonomy. e.g. 'token.consumed'
timestamp	ISO 8601 ms	Millisecond-precision timestamp of when the event occurred (not processed).
source_system	string	Identifier of the system that generated the event. For audit and attribution.
customer_id	UUID	The customer whose consumption this event records.
product_id	UUID	The product being consumed.
entitlement_id	UUID	The entitlement governing this consumption. Validated at ingestion.
session_id	UUID (optional)	Groups related events from a single interaction or workflow.
agent_id	UUID (optional)	The AI agent that initiated this event, if applicable.
payload	JSON object	Type-specific fields. Schema defined by event type.

CHAPTER THREE

The Canonical Objects — Continued

Part 2b — Invoices, Contracts, Credits, Allocations, and the remaining object definitions.

Object 6 — Invoice

The Invoice object is the formal commercial document that requests payment for AI services consumed. It is derived from Event data aggregated according to billing rules, and it must be traceable — every line item on an invoice must be traceable to the specific events that generated it.

Required fields: id (UUID), customer_id (UUID), contract_id (UUID), status (enum: draft | issued | disputed | paid | partially_paid | written_off), issue_date (ISO 8601

date), due_date (ISO 8601 date), currency (ISO 4217), subtotal (decimal), tax_amount (decimal), total_amount (decimal), period_start (ISO 8601 date), period_end (ISO 8601 date), created_at (ISO 8601 timestamp).

Line item structure: each invoice contains an array of line items. Each line item contains: line_id (UUID), product_id (UUID), price_id (UUID), entitlement_id (UUID), description (string), quantity (decimal), unit (string), unit_price (decimal), line_total (decimal), tax_rate (decimal), tax_amount (decimal), event_count (integer, number of events aggregated into this line), event_summary_uri (string, URI to the event-level detail supporting this line).

The event_summary_uri field is a critical traceability mechanism. It provides a pointer to the detailed breakdown of events that generated the line item — a reconciliation report that shows, for each unit charged on the invoice, the specific events from which it was derived. This traceability is what allows a billing dispute to be resolved with evidence rather than assertion: the vendor can show the customer exactly which usage events they are being charged for, and the customer can verify those events against their own system logs.

Adjustment mechanism: when an invoice contains an error, the correction is made through an InvoiceAdjustment object rather than by modifying the original invoice. The InvoiceAdjustment references the original invoice's id, specifies the line item being adjusted, records the original amount and the adjusted amount, and requires an authorization reference (the id of the approval that authorizes the adjustment). This mechanism preserves the original invoice as an immutable record while documenting the correction with a complete audit trail.

Invoice Line Item — Required Fields

Field	Type / Values	Description
`line_id`	UUID	Unique identifier for this line within the invoice.
`product_id`	UUID	Product being billed.
`price_id`	UUID	Price object governing this line's calculation.

entitlement_id	UUID	Entitlement whose consumption is being billed.
quantity	decimal	Billable quantity (tokens, tasks, outcomes, units).
unit	string	Unit of measurement. Must match the Meter's unit field.
unit_price	decimal	Price per unit from the Price object.
line_total	decimal	quantity × unit_price. Pre-tax.
event_count	integer	Number of events aggregated into this line. Must be ≥ 1.
event_summary_uri	string	URI to the event-level reconciliation report for this line.

> ***"Every line item on every invoice must be traceable to the specific events that generated it. This is not a compliance requirement. It is the foundation of customer trust."***

Object 7 — Contract

The Contract object is the legal and commercial agreement between a vendor and a customer. It is the source of truth for entitlement terms, pricing, and the conditions under which services are provided. Every Entitlement derives from a Contract, and every Invoice references the Contract that governs its pricing.

Required fields: id (UUID), vendor_id (UUID), customer_id (UUID), status (enum: draft | pending_signature | active | amended | expired | terminated), effective_date (ISO 8601 date), expiry_date (ISO 8601 date, null for evergreen contracts), created_at (ISO 8601 timestamp), contract_value (decimal, total committed contract value), auto_renew (boolean).

Line items: each contract contains an array of contract line items, each specifying a product, a price, the quantity or commitment, the entitlement terms, and the billing frequency.

Amendment handling: contracts change. Customers add products, reduce commitments, swap models, extend terms. The protocol handles amendments through the ContractAmendment object rather than by modifying the original contract. Each amendment records the original terms and the new terms, the effective date of the change, the impact on billing (prospective versus cumulative catch-up), and the authorization chain. The original contract and all amendments together form the complete commercial record.

The Contract object also carries the critical commercial terms that affect revenue recognition: the performance obligation definitions (what the vendor commits to deliver), the transaction price allocation methodology (how the contract value is allocated across performance obligations), the variable consideration estimate (the best estimate of usage-based charges that will ultimately be billed), and the constraint analysis (the factors that limit how much variable consideration can be included in revenue).

Contract Revenue Recognition Fields

Field	Type / Values	Description
`performance_obligations`	array	List of distinct performance obligations with id, description, and satisfaction criteria.
`transaction_price`	decimal	The total transaction price for revenue allocation purposes.
`allocation_method`	enum	relative_standalone_selling_price | residual | cost_plus
`variable_consideration_estimate`	decimal	Best estimate of usage-based charges for the contract period.

variable_consideration_constraint	string	Narrative description of factors constraining variable consideration recognition.
renewal_type	enum	auto_renew | manual_renewal | evergreen | fixed_term_no_renewal

Object 8 — Credit

The remaining seven canonical objects complete the protocol's object model.

Object 9 — Allocation

The Credit object records a reduction in amount owed, applied at the invoice or line level. Required fields: id, customer_id, invoice_id, amount, currency, reason (enum: billing_error | sla_breach | goodwill | promotional | dispute_resolution), authorized_by (UUID, the approver), applied_at (ISO 8601 timestamp). Every credit must trace to an authorization. Credits cannot be self-applied.

Object 10 — Token Budget

The Allocation object governs the distribution of transaction price across performance obligations in multi-element arrangements. It is the rev rec object — the record of how a contract price is allocated for accounting purposes. Required fields: id, contract_id, total_contract_value, currency, allocation_method (enum: relative_standalone_selling_price | residual | cost_plus), allocation_date. Each allocation contains a components array, each element of which specifies a performance obligation, its standalone selling price, and the portion of contract value allocated to it.

Object 11 — Agent Task

The Token Budget object governs AI consumption at the enterprise level — the financial control that CFOs use to manage AI spending. Required fields: id, customer_id, owner_id (the business unit or team the budget belongs to), total_budget_tokens (integer), total_budget_dollars (decimal), consumed_tokens (integer), consumed_dollars (decimal), period_start, period_end, enforcement_policy (enum: hard_limit | soft_limit_with_alert | soft_limit_with_approval), alert_threshold_percentage (integer).

Object 12 — Outcome

The Agent Task object records a discrete unit of agentic work — a single execution of a defined workflow by an AI agent. Required fields: id, agent_id, workflow_id, customer_id, entitlement_id, status (enum: queued | running | completed | failed | cancelled), started_at, completed_at, tokens_consumed (integer), subtasks (array of child Agent Task ids, for orchestrated multi-agent workflows).

Object 13 — Asset

The Outcome object records a verified business result that triggers a billing event. Required fields: id, customer_id, product_id, outcome_definition_id, status (enum: pending_verification | verified | rejected | disputed), claimed_at, verified_at, verified_by (string, the system or person that verified the outcome), economic_value (decimal, if calculable), billing_event_id (UUID, the billing event triggered by this verified outcome).

The Thirteen Objects — Quick Reference		
Field	**Type / Values**	**Description**
`Product`	Defines what is being sold — type, AI layer, capability description	

Price	Defines how the product is charged — type, unit price, tiers, overage rules	
Entitlement	Defines what the customer has the right to consume — budget, limits, enforcement	
Meter	Defines how consumption is measured — unit, precision, aggregation method	
Event	Records a single commercially significant occurrence — immutable, typed, attributed	
Invoice	Requests payment — traceable to events, supports line and invoice adjustments	
Contract	The commercial agreement — source of entitlement and pricing terms	
Credit	Records a reduction in amount owed — requires authorization, creates audit trail	
Allocation	Distributes transaction price across performance obligations — ASC 606 / IFRS 15	
Token Budget	Governs enterprise AI consumption — the CFO's financial control instrument	
Agent Task	Records a discrete unit of agentic work — links to workflow, entitlement, events	

`Outcome`	Records a verified business result — triggers billing, supports SLA management	
`Asset`	Per-customer inventory of AI capabilities — drives renewals, accurate billing	

PART THREE

The Event Taxonomy

Forty-seven canonical event types. The complete vocabulary of what happens in AI commerce.

CHAPTER FOUR

The Event Taxonomy

Six categories. Forty-seven event types. The atomic signals from which all billing, audit, and intelligence derive.

The event taxonomy is the second foundational component of the Monetization Protocol. Where the object model defines the things that exist in AI commerce, the event taxonomy defines the things that happen. Every commercially significant occurrence in an AI deployment — every token processed, every task completed, every outcome verified, every invoice issued, every payment received — is recorded as a typed event with a canonical schema.

The taxonomy has six top-level categories, each representing a distinct domain of commercial activity. Within each category, events are named using a noun.verb convention that makes their meaning unambiguous: token.consumed, task.completed,

outcome.verified, invoice.issued, payment.received. This naming convention is borrowed from event-driven architecture practice and extended to the commercial domain.

Category 1: Consumption events. These are the raw signals of AI usage — the events that drive billing. token.consumed is the most fundamental: it records the processing of input and output tokens by a foundation model, with the precise counts required for per-token billing. task.completed records the successful execution of an agent workflow. task.failed records an agent workflow that did not complete successfully. outcome.verified records the confirmed delivery of a business result. compute.utilized records the consumption of reserved compute capacity.

Category 2: Commercial events. These record changes in commercial state — new contracts, amendments, cancellations, pricing changes. contract.activated, contract.amended, contract.renewed, contract.expired, contract.terminated. entitlement.activated, entitlement.suspended, entitlement.amended, entitlement.expired. price.updated (with the previous price and the new price for audit purposes).

Category 3: Financial events. These record movements of money and value. invoice.drafted, invoice.issued, invoice.disputed, invoice.paid, invoice.written_off. credit.issued, credit.applied. payment.received, payment.failed, payment.reversed. adjustment.approved, adjustment.applied.

Category 4: Governance events. These record control actions — budget alerts, approvals, escalations, agent huddle convocations. budget.warning (at 80% of token budget consumed), budget.exhausted (at 100%), budget.reset (at period renewal). approval.requested, approval.granted, approval.denied, approval.escalated. agent_huddle.convened, agent_huddle.resolved, agent_huddle.escalated.

Category 5: Agent commerce events. These record commercial activities initiated by or involving AI agents acting as autonomous economic participants. agent.purchase_initiated, agent.price_negotiated, agent.contract_accepted,

agent.task_delegated, agent.payment_authorized. These events introduce a new attribute — initiator_type (enum: human | agent) — that tracks whether a commercial action was taken by a human or by an AI agent.

Category 6: Audit events. These record every access to commercial data, every configuration change, and every exception in the processing pipeline. object.read, object.created, object.updated, object.deleted. configuration.changed. pipeline.error, pipeline.retry, pipeline.dlq (dead letter queue entry for events that failed processing after maximum retries).

The complete event taxonomy contains 47 distinct event types across the six categories. Each type has a defined schema, a defined set of required and optional fields, and a defined set of downstream consequences — which objects it updates, which APIs it may trigger, and which audit trail entries it generates.

Category 1: Consumption Events

Event type	Key payload fields	Triggered by / consequence
`token.consumed`	input_tokens, output_tokens, cached_input_tokens, model_version	AI model processes tokens → updates Entitlement consumption counter → contributes to invoice aggregation
`task.completed`	task_definition_id, task_duration_ms, task_success=true	Agent workflow completes → updates Entitlement task counter → triggers billing event if per_task pricing
`task.failed`	task_definition_id, task_duration_ms, error_code	Workflow fails → does not bill → creates audit entry → may trigger SLA monitoring alert
`outcome.verified`	outcome_definition_id, outcome_value, verification_method	Business result confirmed → triggers billing event → updates SLA delivery counter
`compute.utilized`	capacity_unit, quantity_consumed, reservation_id	Reserved capacity consumed → updates utilization metrics → contributes to invoice if overage pricing active

Category 2: Commercial Events

Event type	Key payload fields	Triggered by / consequence
`contract.activated`	contract_id, customer_id, effective_date	Contract execution complete → triggers entitlement.activated for each

		line item → starts asset tracking
`contract.amended`	contract_id, amendment_id, changed_fields	Contract modified → may trigger new Entitlement objects → updates rev rec Allocation → audit event
`contract.expired`	contract_id, expiry_date	Contract term ends → triggers entitlement.expired → Asset status → pending_renewal
`entitlement.activated`	entitlement_id, customer_id, product_id	Provisioning complete → entitlement.check begins returning permitted=true → consumption tracking starts
`entitlement.budget_warning`	entitlement_id, consumed_percentage=0.80	80% of token budget consumed → notification to customer → governance alert if policy configured
`entitlement.budget_exhausted`	entitlement_id, consumed_percentage=1.00	100% of token budget consumed → if hard_limit: entitlement.check returns permitted=false
`price.updated`	price_id, previous_price, new_price, effective_from	Price change recorded → audit trail → future billings use new price, historical billings unchanged

Category 3: Financial Events

Event type	Key payload fields	Triggered by / consequence
`invoice.issued`	invoice_id, customer_id, total_amount, due_date	Invoice sent to customer → payment window opens → DSO clock starts
`invoice.disputed`	invoice_id, dispute_id, disputed_amount	Dispute filed → automated resolution attempted → Agent Huddle triggered if unresolved
`invoice.paid`	invoice_id, payment_id, amount_received	Payment confirmed → revenue recognized → AR balance updated
`credit.applied`	credit_id, invoice_id, amount	Credit applied → invoice balance reduced → revenue recognition adjusted
`adjustment.applied`	adjustment_id, invoice_id, amount, reason_code	Billing adjustment recorded → both original and adjusted amounts preserved for audit

Category 4: Governance Events

Event type	Key payload fields	Triggered by / consequence

budget.warning	token_budget_id, consumed_percentage, projected_exhaustion_date	Consumption tracking → CFO dashboard update → notification to budget owner
budget.exhausted	token_budget_id, period_end_date	Hard limit reached → new consumption requests rejected until reset or increase → alert to CFO
approval.requested	approval_id, request_type, requester_id, approver_id, amount	Adjustment or budget increase needs authorization → routes to configured approver → SLA timer starts
approval.granted	approval_id, approver_id, timestamp	Authorization confirmed → requested action proceeds → audit trail completed
agent_huddle.convened	huddle_id, huddle_type, participants, context_object_id	Exception requires human+agent resolution → participants notified → resolution timer starts
agent_huddle.resolved	huddle_id, resolution, authorized_by	Huddle reaches decision → authorized actions executed by agents → decision recorded in audit log

Category 5: Agent Commerce Events

Event type	Key payload fields	Triggered by / consequence
agent.purchase_initiated	agent_id, product_id, requested_quantity, budget_check_passed	AI agent initiates purchase → validated against agent commercial authority → logged with agent_id
agent.contract_accepted	agent_id, contract_id, terms_hash	Agent accepts commercial terms → contract activated → entitlement provisioned
agent.task_delegated	delegating_agent_id, receiving_agent_id, task_spec, price_agreed	Agent delegates work to another agent → creates sub-entitlement → establishes audit chain
agent.payment_authorized	agent_id, invoice_id, amount, within_authority_limit	Agent authorizes payment within configured limits → payment initiated → audit trail with agent_id

The event processing pipeline is the infrastructure through which raw usage signals are transformed into billable quantities, through which commercial state changes are propagated, and through which audit trails are maintained. Understanding the pipeline is essential for implementing the protocol correctly.

The pipeline has five stages: ingestion, validation, attribution, aggregation, and archival.

Ingestion is the entry point: events arrive at the event store from multiple sources — model inference engines, agent orchestration frameworks, manual billing operations, payment processors. The ingestion layer accepts events in the canonical JSON format defined by the protocol, validates the event structure against the appropriate schema, assigns a processing timestamp, and writes the event to the event store. Events are immutable once written. The ingestion layer rejects malformed events with a detailed error response rather than silently discarding them.

Validation confirms that the event is commercially coherent — not just structurally valid, but semantically valid in the context of the current commercial state. A token.consumed event for a customer with an expired entitlement should not be silently accepted; it should be flagged as an anomaly for resolution. A task.completed event for a task_id that has already been marked completed is a duplicate that requires investigation. Validation is the protocol's defense against billing errors that originate in the event layer.

Attribution maps each event to its billing context: the customer, the product, the price, the entitlement, and the billing period. Attribution failures — events that cannot be mapped to a valid commercial context — are the most common source of revenue leakage. The protocol requires that attribution failures be placed in a resolution queue where they are reviewed within 24 hours. An event that cannot be attributed within 72 hours triggers a protocol-level alert.

Aggregation collects attributed events and computes billable quantities according to the pricing rules defined in the Price and Meter objects. Token events are summed within billing periods and multiplied by the applicable token prices. Task events are counted and multiplied by task prices. Aggregation runs continuously for real-time billing visibility and runs definitively at the billing period boundary to generate invoice line items.

Archival maintains the complete, immutable event history for the retention period required by applicable accounting standards and the customer's contract — typically seven years minimum for revenue-affecting events. The archive must be queryable: a

customer who disputes an invoice must be able to access the specific events that generated the disputed charges within 48 hours.

PART FOUR

The Standard APIs

Six API groups. The interfaces through which any system can price, meter, bill, and settle.

CHAPTER FIVE

The Catalog and Entitlement APIs

Service discovery, product details, pricing intelligence, and consumption governance.

The standard APIs are the interfaces through which systems that implement the Monetization Protocol interact. They define the operations — the verbs — that any compliant system must expose and any compliant client must be able to invoke.

The API design follows REST principles with some modifications for the specific requirements of commercial AI infrastructure. Resources are identified by URIs. Operations are expressed through HTTP methods. Responses are in JSON. Authentication uses OAuth 2.0 with scopes that map to the protocol's permission model. All API endpoints require TLS 1.3 or higher. Rate limits are expressed in the response headers using standard rate limit headers.

The protocol defines six API groups, each corresponding to a domain of commercial activity: the Catalog API for product and price management, the Entitlement API for consumption governance, the Metering API for usage recording, the Billing API for invoice and payment management, the Settlement API for financial settlement, and the Intelligence API for analytics and reporting.

Each API is defined with: the endpoint URI pattern, the supported HTTP methods, the required request headers, the request body schema, the response schema, the error codes, and the rate limit tier. The complete API reference is provided in Appendix A.

Catalog API

The Catalog API manages the commercial offering — the products, prices, and bundles that are available for sale.

GET /v1/products returns a paginated list of active products. Query parameters: type (filter by product type), ai_layer (filter by AI economy layer), status (default: active), limit (default: 20, max: 100), cursor (for pagination). Response: array of Product objects, with pagination metadata.

GET	/v1/products	Returns paginated list of active products
Parameters / Request body:		
type	*optional*	Filter by product type enum value
ai_layer	*optional*	Filter by AI economy layer (compute/model/token/agent/outcome)
limit	*optional*	Results per page. Default 20, max 100
cursor	*optional*	Pagination cursor from previous response
Response: Array of Product objects with pagination metadata (total_count, next_cursor)		

POST	/v1/products	Creates a new product in draft status
Parameters / Request body:		
name	*required*	Human-readable product name
type	*required*	Product type enum value
ai_layer	*recommended*	The AI economy layer this product operates at
capability_description	*recommended*	Natural language description for agent discovery
Response: Created Product object with assigned id and status: draft		

GET	/v1/catalog/search	Semantic search across catalog — designed for AI agent use
Parameters / Request body:		
query	*required*	Natural language description of the capability needed
budget_per_unit	*optional*	Maximum price per unit the requesting agent will pay
ai_layer	*optional*	Filter by layer
limit	*optional*	Maximum results to return. Default 5.
Response: Array of ranked matches: product_id, price_id, match_score, capability_summary, pricing_summary		

GET	/v1/pricing/estimate	Returns projected cost for a consumption estimate
Parameters / Request body:		
product_id	*required*	Product to price
estimated_consumption	*required*	Object: {type: string, quantity: decimal, period: enum}
customer_id	*optional*	If provided, applies customer-specific pricing or discounts
Response: Projected cost with confidence interval, breakdown by pricing component, vs. current spend comparison		

Entitlement API

The Entitlement API manages the consumption rights that customers have purchased. It is the primary interface for access control and governance.

POST /v1/entitlements/check is the most performance-critical endpoint in the protocol. It is called on every AI request to determine whether the requesting customer has an active entitlement that covers the requested consumption. Request body: customer_id, product_id, requested_quantity (optional, the amount of consumption being requested), request_type (enum: authorize | consume). Response: allowed (boolean), entitlement_id (UUID, if allowed), remaining_budget (integer or null), enforcement_policy (enum: hard_limit | soft_limit).

POST	/v1/entitlements/check	Verifies consumption permission. Most performance-critical endpoint.
Parameters / Request body:		
customer_id	*required*	Customer requesting consumption
product_id	*required*	Product being consumed
requested_quantity	*optional*	Amount of consumption to authorize. If omitted, checks existence only.
request_type	*required*	authorize (check only) | consume (check and record atomically)
Response: allowed (boolean), entitlement_id, remaining_budget, enforcement_policy. P99 latency target: 20ms		

PATCH	/v1/entitlements/{entitlement_id}	Updates mutable entitlement fields
Parameters / Request body:		
status	*optional*	active | suspended — for manual governance actions
token_budget	*optional*	Increased budget amount — requires authorization_reference
authorization_reference	*conditional*	Required when changing token_budget. References approval event id.
Response: Updated Entitlement object		

POST	/v1/entitlements/{entitlement_id}/reset	Resets consumption counter at period boundary
Parameters / Request body:		
authorization_reference	*required*	Must reference a valid governance event authorizing the reset

Response: Updated Entitlement with reset consumption counters. Generates entitlement.reset governance event.	

CHAPTER SIX

The Metering, Billing, and Settlement APIs

Recording consumption, generating invoices, processing payments, and settling accounts.

Metering API

The Metering API is the interface through which AI systems record consumption. It must support extremely high throughput — an enterprise AI deployment may generate millions of metering events per hour — and it must be reliable under the failure modes that are common in distributed AI infrastructure.

POST /v1/events is the primary consumption recording endpoint. Request body: a single Event object or an array of Event objects (batch mode). The API accepts batches of up to 1,000 events per request. Response: array of accepted event ids, array of rejected events with error codes (for validation failures within a batch).

POST	/v1/events	Records consumption events. Supports single event or batch of up to 1,000.
Parameters / Request body:		
events	*required*	Event object or array of Event objects. Each must have globally unique id.
x-idempotency-key	*optional*	Header. If provided, duplicate submissions with same key return original result.
Response: Array of {event_id, status: accepted|rejected, error (if rejected)}. Partial success is valid.		

POST	/v1/events/bulk	Asynchronous bulk event ingestion for large datasets

Parameters / Request body:		
`file`	*required*	JSON Lines file. One Event object per line. Multipart upload.
`customer_id`	*required*	Must match customer_id in all events in the file.

Response: {job_id, status: processing, estimated_completion_at}. Poll GET /v1/events/bulk/{job_id} for status.

GET	`/v1/events`	Queries the event store. Primary interface for billing reconciliation.
Parameters / Request body:		
`customer_id`	*required*	Filter to specific customer
`type`	*optional*	Filter by event type
`from`	*optional*	ISO 8601 timestamp — start of range
`to`	*optional*	ISO 8601 timestamp — end of range
`session_id`	*optional*	Filter to events from a specific interaction session

Response: Paginated array of Event objects. Maximum 10,000 per request. Use cursor for pagination.

GET	`/v1/events/{event_id}/attribution`	Traces an event through the golden thread to its invoice
Parameters / Request body:		
`event_id`	*required*	Path parameter — the event to trace

Response: {event, customer, product, entitlement, billing_period, aggregation_batch_id, invoice_id, line_item_id}

Billing API

The Billing API manages the invoice lifecycle — from draft generation through issuance, dispute resolution, and payment.

POST /v1/billing/run triggers the billing calculation for a specified customer and billing period. This endpoint aggregates all events in the specified period, applies the applicable pricing rules, computes taxes, and creates a draft invoice. Request body: customer_id, period_start, period_end, dry_run (boolean, if true returns the invoice calculation

without creating a draft). Response: draft Invoice object or, for dry_run, the computed invoice with calculations but no id assigned.

POST	/v1/billing/run	Triggers billing calculation for a customer and period
Parameters / Request body:		
customer_id	*required*	Customer to bill
period_start	*required*	ISO 8601 date — start of billing period
period_end	*required*	ISO 8601 date — end of billing period
dry_run	*optional*	Boolean. If true, returns calculation without creating invoice.
Response: Draft Invoice object (status: draft) or, for dry_run, Invoice-shaped calculation object without id.		

POST	/v1/invoices/{invoice_id}/dispute	Opens a billing dispute
Parameters / Request body:		
reason_code	*required*	Enum: billing_error | sla_breach | duplicate_charge | unauthorized_charge | other
disputed_line_items	*required*	Array of line_item ids being disputed
customer_description	*required*	Customer's description of the dispute
supporting_evidence	*optional*	Array of URIs to supporting documents
Response: {dispute_id, status: open, tier_1_resolution_attempted: boolean, expected_resolution_days, actions_taken[]}		

GET	/v1/invoices/{id}/line_items/{line_id}/events	Event-level detail for a line item
Response: Reconciliation report: {line_item, event_count, events[], aggregation_steps[], pricing_calculation}		

Settlement API

The Settlement API handles the financial settlement of invoices — the movement of money from customer to vendor.

POST /v1/payments records receipt of a payment. This endpoint is called when a payment is confirmed by the payment processor. Request body: invoice_id, amount, currency, payment_method, payment_date, reference_number. Response: Payment object with status (pending until confirmed by reconciliation).

POST	/v1/settlement/agent	Grants payment authority to an AI agent
Parameters / Request body:		
agent_id	*required*	The Agent Identity id of the agent being authorized
payment_authority	*required*	Decimal — maximum single payment the agent can authorize
period_limit	*required*	Decimal — maximum total payments per billing period
requires_human_confirmation_above	*optional*	Decimal — threshold requiring human approval
Response: {authorization_id, agent_id, granted_at, payment_authority, period_limit, confirmation_threshold}		

GET	/v1/settlement/position	Current financial position for a customer
Parameters / Request body:		
customer_id	*required*	Customer to assess
as_of	*optional*	ISO 8601 date. Defaults to current date.
Response: {total_outstanding, total_payments, unapplied_credits, net_amount_due, aging_buckets[], currency}		

PART FIVE

Agent Interfaces

How AI agents participate in the commercial economy as buyers, sellers, and brokers.

CHAPTER SEVEN

Agent-to-Agent Commerce

The protocol extensions that enable AI agents to discover, negotiate, transact, and settle — autonomously, auditably, and within enterprise governance frameworks.

Agent-to-agent commerce is the frontier of the Monetization Protocol — the most forward-looking component and the one that will determine whether the protocol can evolve with the AI economy rather than becoming obsolete as AI agents become increasingly autonomous economic actors.

The premise is straightforward but its implications are profound: AI agents are increasingly capable of performing commercial tasks autonomously — purchasing services, negotiating terms, verifying delivery, authorizing payment. When one AI agent can hire another AI agent to perform a task, the commercial transaction between them requires the same elements as any commercial transaction: a description of what is being purchased, a price, a payment mechanism, and a record of delivery. If these elements are not standardized, every agent-to-agent commerce implementation becomes a bespoke bilateral agreement that cannot scale.

The agent commerce extensions to the Monetization Protocol define four capabilities: service discovery (how an agent finds and evaluates commercial AI services), negotiation (how an agent agrees terms with a service provider), execution and monitoring (how an agent purchases, monitors, and receives delivery of AI services), and settlement (how an agent confirms delivery and authorizes payment).

These extensions are deliberately minimal. The goal is not to specify every aspect of agent commerce — that would require solving problems that are not yet well understood. The goal is to specify the minimum viable set of interfaces that allows agent commerce to function at scale, in a way that is auditable, governable, and compatible with enterprise financial controls.

> ***"When one AI agent can hire another AI agent, the commercial transaction between them requires the same elements as any commercial transaction. If those elements are not standardized, every agent-to-agent implementation becomes a bespoke bilateral agreement that cannot scale."***

Service Discovery

Service discovery is the process by which an AI agent finds AI services that meet its requirements. In human-driven commerce, service discovery relies on search engines, recommendations, and prior relationships. In agent commerce, it requires a machine-readable service catalog with semantically queryable capabilities.

The Catalog API's GET /v1/catalog/search endpoint provides the primary discovery interface. But the protocol also defines a structured capability advertisement format that service providers use to describe their offerings in terms that are semantically meaningful to purchasing agents.

The Capability Advertisement object contains: id (UUID), product_id (UUID), name (string, human-readable name), description (string, detailed natural language description of the capability), tags (array of standardized capability tags from the protocol's tag taxonomy), input_format (object, describing what the service accepts),

output_format (object, describing what the service delivers), performance_characteristics (object, including average_latency_ms, throughput_rate, reliability_percentage), pricing_summary (object, including type and indicative price range), availability (object, including rate limits and geographic availability), sample_io (array of anonymized input-output examples).

The tag taxonomy is a controlled vocabulary of capability descriptors that allows agents to search for services by capability type without relying on natural language similarity. Tags include: NLP:summarization, NLP:translation, NLP:classification, Code:generation, Code:review, Code:testing, Document:extraction, Document:analysis, Research:web_search, Research:synthesis, Domain:legal, Domain:finance, Domain:healthcare, and several dozen others. A service provider tags their capability advertisement with all applicable tags, and a purchasing agent can search for services by tag with deterministic results rather than relying on probabilistic semantic search.

The discovery protocol also defines a challenge-response verification mechanism: a purchasing agent can submit a standardized test query to a service provider and compare the response against a reference answer to verify that the service performs as advertised before committing to a purchase. This reduces information asymmetry in agent commerce and creates accountability for capability claims.

Capability Advertisement Object — Key Fields

Field	Type / Values	Description
`id`	UUID	Stable identifier for this capability advertisement
`product_id`	UUID	Reference to the Product object in the Catalog
`tags`	array	Controlled vocabulary tags from the protocol's tag taxonomy. Machine-searchable.
`input_format`	object	Schema describing what the service accepts as input
`output_format`	object	Schema describing what the service delivers as output

performance_characteristics	object	avg_latency_ms, throughput_rate, reliability_percentage
sla_commitments	array	Machine-readable SLA specifications (see SLA format below)
sample_io	array	Anonymized input-output pairs for capability verification
pricing_summary	object	type (enum), indicative_price_range {min, max, unit}

Capability Tag Taxonomy — Selected Categories

Field	Type / Values	Description
NLP:summarization	Condenses long text into shorter summaries	
NLP:translation	Translates text between languages	
NLP:extraction	Extracts structured data from unstructured text	
Code:generation	Writes new code from specifications or descriptions	
Code:review	Reviews code for quality, bugs, security issues	
Code:testing	Generates test cases and test data	
Document:analysis	Analyzes documents for content, structure, sentiment	
Research:web_search	Searches the web and synthesizes findings	
Research:synthesis	Synthesizes information from multiple sources	
Domain:legal	Legal document analysis, contract review, compliance checking	

`Domain:finance`	Financial analysis, forecasting, reporting	
`Domain:healthcare`	Clinical decision support, medical record analysis	
`Agentic:orchestration`	Coordinates multiple agents to complete complex workflows	
`Agentic:long_running`	Executes workflows that span hours or days	

Negotiation Protocol

Commercial negotiation between AI agents is currently unspecified territory. Most agent-to-agent transactions today use fixed pricing — the service provider publishes a price and the purchasing agent either accepts it or does not. But as agent commerce matures, dynamic pricing, volume commitments, and SLA customization will require a negotiation mechanism.

The protocol defines a lightweight negotiation protocol based on offer and acceptance, analogous to how HTTP defines request and response. The negotiation protocol is intentionally simple — it is not designed to support complex multi-round negotiations, but to provide a standard mechanism for the most common agent commerce negotiation patterns.

A purchase_intent request is the opening message: the purchasing agent sends its requirements to the service provider. The request contains: capability_tags (the capabilities needed), input_description (what the agent will provide as input), output_requirements (what the agent requires as output), volume_estimate (optional, anticipated volume over the proposed contract period), timeline (when the service is needed), budget_per_unit (optional, the agent's budget constraint), sla_requirements (optional, required performance characteristics).

A purchase_offer response is the service provider's response to a purchase_intent. It contains: offer_id (UUID, valid for the duration specified), product_id, price_id, quantity_unit, price_per_unit, volume_discounts (optional tiers), sla_commitments (the SLA the provider commits to for this transaction), contract_terms (reference to the standard terms that govern the transaction), expiry (timestamp after which the offer is no longer valid).

An offer_acceptance confirms the terms. An offer_rejection declines the terms, optionally with a counter-offer. A contract_executed event records that both parties have accepted terms and the service relationship has commenced.

The negotiation protocol includes a machine-readable SLA specification format that defines performance commitments in terms that can be monitored programmatically. An SLA specification contains: metric (string, the performance metric being committed to, from a standard taxonomy: latency_p50_ms, latency_p99_ms, availability_percentage, task_completion_rate, outcome_delivery_rate), threshold (decimal), measurement_period (enum: rolling_1h | rolling_24h | monthly), remedy (object, specifying what credits or other remedies apply if the SLA is breached).

The machine-readable SLA is one of the protocol's most operationally significant innovations. When SLA commitments are expressed in a standard format that both the service provider and the purchasing agent can process programmatically, SLA monitoring and breach detection can be automated. The purchasing agent can monitor the service against its commitments without human involvement, and the credit calculation for SLA breaches can be applied automatically without a dispute process.

Machine-Readable SLA Specification Format

Field	Type / Values	Description
`metric`	string	Performance metric from standard taxonomy: latency_p50_ms | latency_p99_ms | availability_percentage | task_completion_rate | outcome_delivery_rate

threshold	decimal	The committed performance level. e.g. 99.9 for availability
measurement_period	enum	rolling_1h | rolling_24h | monthly
remedy_type	enum	credit_percentage | fixed_credit | service_extension | refund
remedy_calculation	string	Formula for remedy. e.g. '10% of monthly_invoice for each 0.1% below threshold'
maximum_remedy	decimal	Cap on total remedy per measurement period as percentage of invoice
measurement_source	string	System providing the performance metrics for SLA assessment

Agent Identity and Commercial Authority

Agent identity is the foundational trust problem in agent commerce. When an AI agent initiates a purchase, the service provider needs to know: who is responsible for this transaction? What organization does this agent represent? What payment authority does it have? Is this agent authorized to make this type of purchase?

The protocol defines an Agent Identity object that establishes the commercial identity of an AI agent in a way that is verifiable, auditable, and compatible with enterprise authorization frameworks.

Required fields: id (UUID, the agent's stable identity across all transactions), display_name (string), parent_organization_id (UUID, the organization that owns and operates this agent), agent_type (string, describing the agent's function), created_at (ISO 8601 timestamp), public_key (string, the agent's public key for cryptographic verification of signed requests).

Commercial authority fields: payment_authority_limit (decimal, maximum single transaction the agent can authorize), period_payment_limit (decimal, maximum payments per billing period), requires_human_confirmation_above (decimal, threshold requiring human confirmation), approved_product_categories (array of

product category tags the agent is authorized to purchase), approved_vendors (array of vendor ids the agent is authorized to transact with, null for unrestricted).

Every API request made by an AI agent includes an Authorization header with a JWT (JSON Web Token) signed by the agent's private key. The JWT payload includes the agent_id, the requesting organization's id, a timestamp, and a nonce. Service providers validate the JWT signature against the agent's registered public key, verify that the requested transaction is within the agent's commercial authority, and record the agent_id in the transaction audit trail.

The agent identity system integrates with enterprise IAM (Identity and Access Management) frameworks through a standard delegation mechanism: an enterprise administrator creates an Agent Identity object, configures its commercial authority limits, and the agent's JWT-signed requests are automatically validated against those limits without requiring human review of individual transactions. This enables fully automated agent commerce within pre-approved boundaries — the enterprise CFO defines the boundaries, the agents operate within them, and every transaction is recorded with the agent_id for audit purposes.

Agent Identity Object — Commercial Authority Fields

Field	Type / Values	Description
`id`	UUID	Stable identity across all transactions. Registered by parent organization.
`parent_organization_id`	UUID	The enterprise that owns and operates this agent. Financial responsibility.
`public_key`	string	Agent's public key for cryptographic verification of signed requests.
`payment_authority_limit`	decimal	Maximum single transaction the agent can authorize without human approval.

`period_payment_limit`	decimal	Maximum total payments the agent can authorize per billing period.
`requires_human_confirmation_above`	decimal	Threshold above which human approval is required.
`approved_product_categories`	array	Capability tags the agent is authorized to purchase. null = unrestricted.
`approved_vendors`	array	Vendor ids the agent is authorized to transact with. null = unrestricted.

PART SIX

The MCP Monetization Server

A complete implementation of the Monetization Protocol as a Model Context Protocol server.

CHAPTER EIGHT

MCP Server Specification

Fifteen tools. Six resources. Twelve prompts. The complete specification for making any MCP-capable AI agent commercially capable.

The Model Context Protocol (MCP), developed by Anthropic, defines a standard interface for providing AI models with access to tools, resources, and prompts. An AI model equipped with MCP can discover available tools, invoke those tools, and receive their results as part of its context — enabling it to take actions in the world rather than merely generating text.

The Monetization Protocol's MCP server implementation makes the commercial capabilities of the Monetization Protocol directly accessible to any MCP-capable AI

agent. An agent connected to the Monetization Protocol MCP server can price an AI service, check a customer's entitlement, record consumption, generate an invoice, initiate a payment, and investigate a billing dispute — all through natural language instructions, without requiring the agent to know the specifics of the underlying API.

This is a profound commercial capability. It means that any AI agent — not just agents specifically built for commercial tasks — can participate in the AI economy's commercial infrastructure. A general-purpose AI assistant can help a procurement manager compare AI service offerings, evaluate their pricing relative to budget, and initiate a purchase. An AI coding agent can check its own token budget before starting a long task and request a budget increase if needed. An AI analytics agent can investigate an anomalous billing event and initiate a dispute if it identifies a billing error.

The MCP server specification in this chapter is a complete, implementable specification. It defines every tool, every resource, and every prompt that the server exposes, with the input schemas, output schemas, and behavioral descriptions required for a conformant implementation.

> ***"The MCP monetization server means any AI agent — not just agents built for commercial tasks — can participate in the AI economy's commercial infrastructure."***

The Fifteen Tools

The Monetization Protocol MCP server exposes fifteen tools, organized into four groups: catalog tools, entitlement tools, billing tools, and commercial intelligence tools.

Catalog tools:

search_catalog accepts a natural language description of an AI capability and returns matching products and prices. Input: query (string), budget_per_unit (optional

decimal), ai_layer (optional enum). Output: array of matches with product details, pricing, and capability summaries. This is the agent's primary tool for service discovery — an agent can describe what it needs in plain language and receive a structured list of available services with their commercial terms.

Catalog Tools — Reference

Field	Type / Values	Description
search_catalog	NL query → ranked product matches	Input: query, budget_per_unit?, ai_layer? | Output: [{product_id, price_id, match_score, summary}]
get_product	product_id → full product details	Input: product_id | Output: complete Product object with all fields
compare_products	product_ids[] → structured comparison	Input: product_ids[] | Output: side-by-side comparison + recommendation summary
get_pricing_estimate	product_id + consumption → projected cost	Input: product_id, estimated_consumption, period | Output: cost projection with confidence interval

Entitlement Tools — Reference

Field	Type / Values	Description
check_entitlement	customer+product → permission check	Input: customer_id, product_id, requested_consumption? | Output: permitted, entitlement_id, remaining_budget, days_to_expiry
get_budget_status	customer → token budget position	Input: customer_id, scope? | Output: [{budget_id, utilization%, remaining, burn_rate, projected_exhaustion_date}]
request_budget_increase	entitlement_id + amount → approval workflow	Input: entitlement_id, requested_increase, justification | Output: request_id, approval_required, estimated_approval_time_hours

Billing Tools — Reference

Field	Type / Values	Description
get_invoice	invoice_id → full invoice	Input: invoice_id | Output: Invoice with all line items and summary statistics

explain_invoice	invoice_id → plain-language explanation	Input: invoice_id | Output: narrative explanation + structured charge breakdown
investigate_charge	line_item_id → audit trail	Input: line_item_id | Output: step-by-step trace: events → aggregation → pricing → invoice
initiate_dispute	invoice + lines → dispute opened	Input: invoice_id, disputed_line_item_ids[], reason_code, description | Output: dispute_id, resolution_timeline, actions_taken[]

Commercial Intelligence Tools — Reference

Field	Type / Values	Description
analyze_spending	customer + period → spending analysis	Input: customer_id, period_start, period_end, breakdown_by | Output: spending summary, trends, anomalies, recommendations
forecast_consumption	customer + horizon → spend forecast	Input: customer_id, forecast_horizon_months, growth_assumption? | Output: monthly forecast, confidence intervals, scenario analysis
detect_leakage	customer + period → leakage report	Input: customer_id, period_start, period_end | Output: total leakage, category breakdown, specific anomalies, resolutions
compare_vendors	vendor names + use case → evaluation	Input: vendor_names[], use_case, estimated_monthly_consumption | Output: structured comparison, TCO analysis, recommendation

The Six Resources

The MCP server exposes six resources — data sources that an AI agent can read to inform its commercial decisions and operations.

monetization_protocol_reference: The complete text of the Monetization Protocol specification, formatted for AI consumption. An agent that needs to understand a specific protocol concept — how variable consideration works under ASC 606, what the valid lifecycle states for an entitlement are, how a composite price is structured — can access this reference to get precise, authoritative answers.

product_catalog: The current product catalog for the implementing organization, including all active products, their prices, and their availability. This resource is kept

current in real time — an agent reading the catalog is reading the live commercial state of the organization's AI offerings.

customer_commercial_state: The complete commercial state for the current customer context — all active entitlements, current token budgets, open invoices, payment history, and pending disputes. This resource is scoped to the authenticated customer and provides a unified view of their commercial relationship with the vendor.

billing_health_index: The current Billing Health Index scores and component metrics for the organization. An agent with access to this resource can assess the overall quality of the organization's billing operations, identify the components with the lowest scores, and prioritize improvement actions.

usage_adoption_score: The current Usage Adoption Score and component metrics for each customer. An agent monitoring customer health can use this resource to identify customers at risk of churn, customers with expansion potential, and customers where usage is not matching the pattern expected for their product tier.

vendor_landscape: A curated reference on the AI vendor ecosystem — product categories, representative vendors, typical pricing ranges, and SLA benchmarks. This resource enables intelligent vendor comparison and market-rate assessment without requiring internet access.

MCP Resources — Reference

Field	Type / Values	Description
`monetization_protocol_reference`	Complete protocol specification formatted for AI consumption	Updated with each protocol version. Authoritative source for protocol questions.
`product_catalog`	Live product catalog for the implementing organization	Real-time product and price data. Scoped to the authenticated vendor context.
`customer_commercial_state`	Complete commercial state	Entitlements, budgets, invoices, payment history, disputes.

	for the current customer	Scoped to authenticated customer.
billing_health_index	Current BHI scores and component metrics	Updated daily. Shows overall billing quality and component breakdown.
usage_adoption_score	Current UAS scores by customer	Updated weekly. Leading indicator of renewal risk and expansion potential.
vendor_landscape	Curated AI vendor ecosystem reference	Product categories, representative vendors, typical pricing ranges, SLA benchmarks.

The Twelve Prompts

The MCP server defines twelve prompt templates — pre-built instructions that configure an AI agent for specific monetization-related tasks. Prompts are the MCP equivalent of a job description: they tell an agent what it is supposed to do, what tools to use, and what output to produce.

analyze_billing_dispute: Configures an agent to investigate a billing dispute. The agent is instructed to: retrieve the disputed invoice, explain each disputed charge in plain language, trace the charges to their underlying events, identify any discrepancies between the events and the billing calculation, calculate the value of any identified discrepancies, draft a dispute response with supporting evidence, and recommend a resolution. The prompt emphasizes accuracy and evidence — the agent should not suggest a resolution without supporting it with specific event-level data.

prepare_cfo_briefing: Configures an agent to prepare a CFO-ready briefing on AI spending. The agent is instructed to: retrieve current spending across all AI products, compare spending to budget, identify the top five cost drivers, flag anomalies, project spending through the end of the period, and format the output as a concise briefing with key numbers prominently displayed and supporting detail in an appendix.

optimize_token_budget: Configures an agent to analyze a customer's token consumption patterns and recommend budget optimizations. The agent is instructed to: retrieve consumption by team, workflow, and model; identify the highest-cost workflows; analyze whether those workflows are achieving their intended outcomes; identify opportunities for prompt optimization, model downgrading, or caching that could reduce consumption without reducing effectiveness; and produce a prioritized list of recommendations with estimated savings.

conduct_vendor_evaluation: Configures an agent to conduct a formal AI vendor evaluation. The agent is instructed to: retrieve the requirements specification, search the vendor landscape for matching capabilities, request pricing estimates from matching vendors using the Catalog API, compare capabilities and pricing against requirements, analyze total cost of ownership, assess SLA terms, and produce an evaluation report with a clear recommendation.

investigate_leakage: Configures an agent to conduct a revenue leakage investigation. The agent is instructed to: run the leakage detection analysis for the specified period, categorize identified leakage by type and amount, trace the highest-value leakage items to their root cause, estimate the total revenue impact, recommend remediation actions for each category, and produce a remediation plan with effort estimates and expected revenue recovery.

monitor_agent_spend: Configures an agent to continuously monitor AI agent spending within an enterprise and alert on anomalies. The agent is instructed to: check token budget utilization across all active agent entitlements on a configured schedule, flag any agent whose consumption rate is tracking to exceed its budget before period end, flag any single-session token consumption that exceeds a configurable threshold, and initiate the budget review workflow for flagged cases.

The six additional prompts cover: generate_renewal_analysis (preparing renewal recommendations for the CS team), reconcile_billing_period (end-of-period billing reconciliation), prepare_audit_pack (assembling revenue recognition evidence for external audit), analyze_partner_commissions (commission calculation and dispute

analysis), benchmark_ai_costs (comparing AI spending against industry benchmarks), and design_token_governance (advising on token budget structure and governance for a new AI deployment).

MCP Prompts — Reference		
Field	**Type / Values**	**Description**
analyze_billing_dispute	Investigate dispute → gather evidence → recommend resolution	Tools: get_invoice, investigate_charge, explain_invoice, initiate_dispute
prepare_cfo_briefing	Current AI spending → board-ready summary	Tools: analyze_spending, get_budget_status, forecast_consumption
optimize_token_budget	Consumption analysis → cost reduction recommendations	Tools: analyze_spending, detect_leakage, get_budget_status
conduct_vendor_evaluation	Requirements → vendor comparison → recommendation	Tools: search_catalog, compare_vendors, get_pricing_estimate
investigate_leakage	Event audit → leakage identification → remediation plan	Tools: detect_leakage, investigate_charge, analyze_spending
monitor_agent_spend	Continuous budget monitoring → anomaly alerting	Tools: check_entitlement, get_budget_status, request_budget_increase
generate_renewal_analysis	Health data → renewal recommendations for CS team	Resources: usage_adoption_score, customer_commercial_state
reconcile_billing_period	Period-end event audit → billing accuracy verification	Tools: detect_leakage, investigate_charge, analyze_spending
prepare_audit_pack	Revenue evidence assembly for external audit	Tools: get_invoice, investigate_charge · Resources: billing_health_index

`analyze_partner_commissions`	Commission calculation audit and dispute analysis	Tools: analyze_spending, investigate_charge
`benchmark_ai_costs`	Spending vs industry benchmarks	Resources: vendor_landscape · Tools: analyze_spending
`design_token_governance`	New deployment → token budget structure advice	Tools: forecast_consumption · Resources: monetization_protocol_reference

PART SEVEN

Governance and Compliance

Dispute resolution, compliance certification, protocol evolution, and participant accountability.

CHAPTER NINE

The Governance Framework

A protocol without governance is a suggestion. The rules for disputes, compliance, evolution, and accountability.

A protocol without governance is a suggestion. The Monetization Protocol's governance framework defines how disputes are resolved, how compliance is assessed, how the protocol evolves, and how participants are held accountable to the standards it establishes.

The governance framework has four components: dispute resolution, compliance certification, protocol evolution, and participant accountability.

Dispute resolution under the protocol follows a tiered approach. Tier one is automated resolution: disputes that can be resolved by comparing event data against billing

calculations are resolved automatically by the dispute resolution API. The system retrieves the disputed charges, traces them to their underlying events, verifies the billing calculation, and issues a credit if a calculation error is found. This tier resolves the majority of billing disputes — those that arise from billing calculation errors, attribution errors, or duplicated charges — without human involvement.

Tier two is the Agent Huddle: disputes that cannot be resolved automatically because they involve judgment — a customer believes a task was not completed satisfactorily, or an outcome was delivered but the customer disputes whether it met the SLA definition — are escalated to an Agent Huddle. The Huddle convenes the relevant human personas (billing ops, customer success, technical account management) and AI agents (billing agent, traceability agent, SLA monitoring agent) around the shared context of the dispute. The Huddle has a defined resolution timeline — 48 hours for standard disputes, 5 business days for complex SLA disputes — and a defined escalation path if the timeline is not met.

Tier three is human arbitration: disputes that cannot be resolved through the Agent Huddle because they involve legal interpretation of contract terms or require commercial judgment beyond the authority of operational staff are escalated to a human arbitration process. The protocol does not specify the arbitration mechanism — it leaves that to the commercial agreement between vendor and customer — but it does require that the complete event audit trail be made available to the arbitrator and that the arbitrator's decision be recorded as a protocol-level event for audit purposes.

Compliance certification establishes the minimum requirements for claiming conformance with the Monetization Protocol. A conformant implementation must implement all required objects with schemas that validate against the protocol's JSON schemas. It must implement all required API endpoints with behavior that conforms to the specification. It must maintain an immutable event store with the required retention period. It must implement the dispute resolution tiers as specified. And it must pass the conformance test suite — a set of automated tests that verify conformant behavior across the full range of specified scenarios.

Protocol evolution follows a structured version management process. The protocol uses semantic versioning: major versions (1.0, 2.0) introduce breaking changes to schemas or APIs; minor versions (1.1, 1.2) add optional fields or new endpoints while maintaining backward compatibility; patch versions (1.0.1) correct errors in the specification without changing behavior. Participants that implement a specific version remain conformant with that version indefinitely — the protocol does not require participants to upgrade. New versions are published with a migration guide and a minimum two-year support window for the previous major version.

Dispute Resolution Tiers

Field	Type / Values	Description
`Tier 1 — Automated`	Billing calculation errors, duplicate charges, attribution errors	Resolution target: 2 hours · Mechanism: automated comparison of event data vs billing calculation · Credit issued automatically if error confirmed
`Tier 2 — Agent Huddle`	SLA disputes, task completion quality, outcome definition disagreements	Resolution target: 48 hours · Mechanism: human + agent Huddle with defined participation and authority · Escalation if timeline missed
`Tier 3 — Human arbitration`	Legal interpretation of contract terms, commercial disputes exceeding Huddle authority	Resolution mechanism: defined by commercial agreement · Complete event audit trail provided to arbitrator · Decision recorded as protocol event

Conformance Requirements — Minimum for Protocol Certification

Field	Type / Values	Description
`Object schemas`	All 13 objects implemented with schemas validating against protocol JSON Schema	Verified by automated conformance test suite
`Required APIs`	All required endpoints implemented with conformant behavior	Verified by API conformance test suite
`Event store`	Immutable event store with minimum 7-year retention	Assessed by architectural review

`Dispute resolution`	All three tiers implemented as specified	Verified by conformance test suite
`Audit trail`	Complete golden thread traceable for all events	Verified by traceability test suite
`Agent interfaces`	Agent Identity, service discovery, and negotiation protocol implemented	Verified by agent commerce conformance tests

Protocol Version Management		
Field	**Type / Values**	**Description**
`Major version (x.0)`	Breaking changes to schemas or APIs	24-month migration window · Previous major version supported for 24 months after new major release
`Minor version (x.y)`	New optional fields, new endpoints	Backward compatible · Previous implementations remain conformant · No migration required
`Patch version (x.y.z)`	Specification corrections, clarifications	No behavioral change · Implementations do not need to update
`Deprecation process`	Features announced deprecated in minor release	Minimum 18 months from deprecation announcement to removal in major release

APPENDIX A

API Quick Reference

All endpoints, organized by API group.

Appendix A contains the complete API reference — every endpoint, every request schema, every response schema, every error code. It is organized by API group: Catalog, Entitlement, Metering, Billing, Settlement, Intelligence, and Agent Commerce. Each endpoint entry includes the URI pattern, supported methods, authentication requirements, request body schema in JSON Schema format, response schema, and example request and response payloads.

Complete API Endpoint Reference		
Field	**Type / Values**	**Description**
`GET /v1/products`	Catalog	List active products. Filter by type, layer, status.
`POST /v1/products`	Catalog	Create product in draft status.
`POST /v1/products/{id}/activate`	Catalog	Transition product to active.
`GET /v1/prices`	Catalog	List prices. Filter by product, type, currency.
`POST /v1/prices`	Catalog	Create new price. Type determines required fields.
`GET /v1/catalog/search`	Catalog	Semantic capability search. Designed for agent use.
`GET /v1/pricing/estimate`	Catalog	Projected cost for consumption estimate.
`POST /v1/entitlements/check`	Entitlement	Check/consume permission. P99 target: 20ms.
`GET /v1/entitlements/{id}`	Entitlement	Retrieve entitlement state and utilization.
`GET /v1/entitlements`	Entitlement	List entitlements for customer.
`PATCH /v1/entitlements/{id}`	Entitlement	Update mutable fields (status, budget).
`POST /v1/entitlements/{id}/reset`	Entitlement	Reset consumption counter at period boundary.
`POST /v1/events`	Metering	Record consumption events (single or batch).
`POST /v1/events/bulk`	Metering	Async bulk event ingestion from JSON Lines file.

`GET /v1/events`	Metering	Query event store. Primary reconciliation interface.
`GET /v1/events/{id}/attribution`	Metering	Trace event through golden thread to invoice.
`POST /v1/billing/run`	Billing	Trigger billing calculation. Supports dry_run.
`GET /v1/invoices/{id}`	Billing	Retrieve complete invoice with line items.
`POST /v1/invoices/{id}/dispute`	Billing	Open billing dispute. Triggers resolution workflow.
`POST /v1/adjustments`	Billing	Create billing adjustment. Requires authorization.
`GET /v1/invoices/{id}/line_items/{lid}/events`	Billing	Event-level detail for line item.
`POST /v1/payments`	Settlement	Record payment receipt.
`POST /v1/payments/apply`	Settlement	Apply payment to invoices.
`GET /v1/settlement/position`	Settlement	Current financial position for customer.
`POST /v1/settlement/agent`	Settlement	Grant payment authority to AI agent.
`GET /v1/intelligence/spending`	Intelligence	Spending analysis by dimension.
`GET /v1/intelligence/forecast`	Intelligence	Consumption and spend forecast.
`GET /v1/intelligence/leakage`	Intelligence	Revenue leakage detection report.
`POST /v1/agents/register`	Agent	Register an Agent Identity object.
`GET /v1/agents/{id}`	Agent	Retrieve agent identity and commercial authority.

POST /v1/commerce/negotiate	Agent	Initiate purchase_intent and receive purchase_offer.
POST /v1/commerce/accept	Agent	Accept a purchase_offer. Creates contract.

APPENDIX B

Event Taxonomy — Complete Reference

All 47 canonical event types.

Appendix B contains the complete event taxonomy — all 47 event types, each with its canonical name, description, required and optional payload fields, and the downstream consequences (object state changes and API triggers) associated with the event.

Event Types — Complete Listing by Category

Field	Type / Values	Description
token.consumed	Category 1: Consumption	input_tokens, output_tokens, cached_input_tokens, model_version, model_id
task.completed	Category 1: Consumption	task_definition_id, task_duration_ms, task_success, workflow_id
task.failed	Category 1: Consumption	task_definition_id, error_code, retry_count, workflow_id
task.partial	Category 1: Consumption	task_definition_id, completion_percentage, reason
outcome.verified	Category 1: Consumption	outcome_definition_id, outcome_value, verification_method, verification_source

outcome.rejected	Category 1: Consumption	outcome_definition_id, rejection_reason, remediation_available
compute.utilized	Category 1: Consumption	capacity_unit, quantity, reservation_id, utilization_percentage
api.called	Category 1: Consumption	endpoint, method, response_time_ms, status_code
contract.activated	Category 2: Commercial	contract_id, vendor_id, customer_id, effective_date
contract.amended	Category 2: Commercial	contract_id, amendment_id, changed_fields[], financial_impact
contract.renewed	Category 2: Commercial	contract_id, new_expiry_date, renewal_terms_changed
contract.expired	Category 2: Commercial	contract_id, expiry_date, renewal_status
contract.terminated	Category 2: Commercial	contract_id, termination_reason, effective_date
entitlement.activated	Category 2: Commercial	entitlement_id, product_id, budget_amount, expiry_date
entitlement.suspended	Category 2: Commercial	entitlement_id, reason_code, reactivation_condition
entitlement.amended	Category 2: Commercial	entitlement_id, changed_fields[], amendment_id
entitlement.budget_warning	Category 2: Commercial	entitlement_id, consumed_percentage, projected_exhaustion_date
entitlement.budget_exhausted	Category 2: Commercial	entitlement_id, enforcement_policy, grace_period_hours
entitlement.expired	Category 2: Commercial	entitlement_id, final_consumed_amount
price.updated	Category 2: Commercial	price_id, previous_price_snapshot, new_price_snapshot, effective_from
invoice.drafted	Category 3: Financial	invoice_id, period_start, period_end, event_count
invoice.issued	Category 3: Financial	invoice_id, customer_id, total_amount, due_date

invoice.disputed	Category 3: Financial	invoice_id, dispute_id, disputed_amount, reason_code
invoice.paid	Category 3: Financial	invoice_id, payment_id, amount_received, payment_date
invoice.written_off	Category 3: Financial	invoice_id, write_off_amount, authorization_reference
credit.issued	Category 3: Financial	credit_id, amount, reason_code, authorized_by
credit.applied	Category 3: Financial	credit_id, invoice_id, amount_applied
payment.received	Category 3: Financial	payment_id, invoice_id, amount, payment_method
payment.failed	Category 3: Financial	payment_id, invoice_id, failure_reason, retry_scheduled
payment.reversed	Category 3: Financial	payment_id, reversal_reason, reversal_date
adjustment.approved	Category 3: Financial	adjustment_id, amount, approver_id, authorization_reference
adjustment.applied	Category 3: Financial	adjustment_id, invoice_id, amount_adjusted
budget.warning	Category 4: Governance	token_budget_id, consumed_percentage, projected_exhaustion_date
budget.exhausted	Category 4: Governance	token_budget_id, consumed_amount, period_end
budget.reset	Category 4: Governance	token_budget_id, new_period_start, reset_amount
budget.increased	Category 4: Governance	token_budget_id, previous_limit, new_limit, authorization_reference
approval.requested	Category 4: Governance	approval_id, request_type, requester_id, approver_id, amount
approval.granted	Category 4: Governance	approval_id, approver_id, approved_at
approval.denied	Category 4: Governance	approval_id, approver_id, denial_reason
approval.escalated	Category 4: Governance	approval_id, from_approver_id, to_approver_id, escalation_reason

agent_huddle.convened	Category 4: Governance	huddle_id, huddle_type, participants[], context_object_id
agent_huddle.resolved	Category 4: Governance	huddle_id, resolution, authorized_by, actions_taken[]
agent_huddle.escalated	Category 4: Governance	huddle_id, escalation_reason, escalated_to
agent.purchase_initiated	Category 5: Agent	agent_id, product_id, requested_quantity, authority_check_passed
agent.contract_accepted	Category 5: Agent	agent_id, contract_id, terms_hash, within_authority_limit
agent.task_delegated	Category 5: Agent	delegating_agent_id, receiving_agent_id, task_spec, price_agreed
agent.payment_authorized	Category 5: Agent	agent_id, invoice_id, amount, authority_remaining

CLOSING

An Invitation to Build

The protocol is a beginning. The work of building it belongs to everyone who will depend on it.

The Monetization Protocol is a beginning, not an end. It is the first complete proposal for what the commercial layer of the AI economy should look like — the first attempt to define the objects, events, APIs, and agent interfaces that would allow any two participants in the AI economy to transact with mutual intelligibility and shared trust.

But protocols only matter if they are adopted. And adoption requires the protocol to be genuinely useful — to solve real problems that real participants in the AI economy are experiencing every day. The integration tax. The semantic inconsistency. The agent incompatibility. The billing disputes that cannot be resolved because the event data is inaccessible or unreliable. The CFO who cannot see AI spending clearly because every vendor represents it differently.

These are solvable problems. The protocol solves them — if it is implemented.

The invitation is open. If you are building AI infrastructure, implement the object model. If you are building a billing system, implement the event taxonomy and the Billing API. If you are building AI agents, implement the agent interfaces and register your agents with the identity framework. If you are building financial governance for an enterprise AI deployment, implement the token budget and FinOps capabilities.

The AI economy is being designed right now. The commercial layer of that economy will either be designed deliberately — with precision, with shared standards, with the rigor that multi-trillion-dollar commercial infrastructure demands — or it will emerge from the accumulation of proprietary, incompatible, bilateral agreements. The first path leads to an economy that can scale, that can be governed, and that can be trusted. The second leads to a commercial infrastructure as fragile and expensive as the legacy banking infrastructure that fintech spent twenty years trying to circumvent.

The protocol is the first path. The work of building it belongs to everyone who will depend on it.

> ***"The AI economy's commercial layer will either be designed deliberately — with precision, with shared standards, with the rigor that multi-trillion-dollar infrastructure demands — or it will emerge from the accumulation of proprietary, incompatible bilateral agreements. The first path leads to scale, governance, and trust. The second leads to fragility. The choice is being made now."***

The AI Economy Monetization Series continues in Book Five:

Monetizing Service as Software in the AI Economy